System for firmware verification

Daniel Nilsson
2009

Kalmar, December 15, 2009
C nivå, 15hp
Datorteknik

Handledare: Torbjörn Holmberg, Ericsson AB

Handledare: Martin Blomberg, Högskolan i Kalmar, Institutionen för kommunikation och design

Examinator: Martin Blomberg, Högskolan i Kalmar Institutionen för kommunikation och design

Institutionen för kommunikation och design Högskolan i Kalmar

Summary

This paper presents research around software verification with emphasis on testing and software for embedded devices, it also presents a testing-framework called Trassel that was designed and implemented during the writing of this paper.

The work done during the writing of this paper was done for Ericsson AB in Kalmar as a part of a larger research project to improve the process of firmware development.

Explanations of concepts inside or connected to the area of software verification and domain specific languages is explained and referenced to in the theory-chapter.

This paper contains a comparison between a few of the available testing-frameworks for C-code for embedded systems.

Domain specific languages and tools embedded in existing general-purpose languages is explained, researched and discussed in this paper. This technique was used for implementing the language Trassel users use to describe test-cases.

While having some rough edges at the end of the project, Trassel is a fully usable testing-framework that allows the user perform automated testing of C-code.

Sammanfattning

Denna rapport visar forskning gällande verifiering av mjukvara med vikt på testning och mjukvara för sk "embedded devices". Den visar även utvecklingen av ett ramverk för testning kallat Trassel som utformades och implementerades under skrivandet av denna rapport.

Arbetet som har utförts under skrivandet av denna rapport har gjorts åt Ericsson AB i Kalmar som en del av ett större forskningsprojekt som ämnar att förbättra processen för firmwareutveckling.

Denna rapport innehåller även förklaringar av koncept underordnade eller relaterade till verifiering av mjukvara och domänspecifika språk och står att finna under "Theory" kapitlet.

Denna rapport jämför även ett fåtal av de tillgängliga ramverken för testning av C-kod i "embedded devices".

Användandet av existerande generella programmeringsspråk för att implementera domänspecifika språk och verktyg förklaras, undersöks och diskuteras i denna rapport. Denna teknik används av Trassel för att implementera ett språk som användare kan nyttja för att beskriva testfall.

Trassel är en ganska oputsad produkt i slutet av arbetet, men den är fullt användbar för att låta utvecklare automatisera testning av C-kod.

Abstract

Software verification is an important part of software development and the most practical way to do this today is through dynamic testing. This report explains concepts connected to verification and testing and also presents the testing-framework Trassel developed during the writing of this report.

Constructing domain specific languages and tools by using an existing language as a starting ground can be a good strategy for solving certain problems, this was tried with Trassel where the description-language for writing test-cases was written as a DSL using Python as the host-language.

Keywords: software verification, unit-testing, domain specific languages, embedded platform, firmware, testing framework.

Contents

1 Introduction **3**
1.1 Purpose/Objective . . . 3
1.2 Limitations . . . 3

2 Theory **5**
2.1 Software verification . . . 5
2.2 Testing . . . 5
2.2.1 Unit-testing . . . 5
2.2.2 Testing-framework . . . 5
2.2.3 Test coverage analysis . . . 6
2.3 FSMs and FSM testing . . . 6
2.4 Endianness (or Byte-ordering) . . . 6
2.5 Software complexity . . . 7
2.6 Domain specific embedded languages . . . 8

3 Method **9**
3.1 Testing frameworks . . . 9
3.2 Implementation languages . . . 10
3.3 Chosen method . . . 11
3.4 Criticism of chosen method . . . 11

4 Realisation **13**
4.1 Design of the Trassel testing-framework . . . 13
4.2 CCS proof-of-concept (Perl) . . . 13
4.3 CCS proof-of-concept (Python) . . . 13
4.4 CCS test runner . . . 14
4.5 GCC test runner . . . 14
4.6 Code-generator modules . . . 14
4.7 Embedded description-language . . . 14
4.8 Generating reports . . . 15
4.9 Automated testing of Trassel . . . 15
4.10 Test-state tree . . . 15
4.11 Application interface . . . 15
4.12 Implementing code-coverage reporting . . . 16
4.13 Implementing test-cases . . . 17

5 Usage example **18**

6 Result **20**
6.1 The testing-framework Trassel . . . 20
6.1.1 Unit-testing . . . 20
6.1.2 Description language . . . 21
6.1.3 Code coverage . . . 21

6.1.4 Documentation 21

7 Analysis / Discussion **22**
7.1 Automated unit-testing 22
7.2 Python 22
7.3 DSEL 22
7.4 Criticism/limitations 23
7.4.1 Limitations inherent in testing 23
7.4.2 Testing method 23
7.4.3 Testing with different platforms 23
7.4.4 Test-case description language 24

8 Conclusion **25**
8.1 Testing 25
8.2 DSELs 25
8.3 Where to go from here 25
8.3.1 Integrate implemented code coverage function 26
8.3.2 Packaging to ease installation and distribution 26
8.3.3 Improving the description language 26
8.3.4 Robustness features 26
8.3.5 Evaluation of alternatives for formal verification 27

A Appendices **29**
A.1 RPN calculator source-code 29
A.1.1 rpn.h 29
A.1.2 rpn.c 29

List of Figures

1 UCD3K DC/DC-converter 3
2 Endianness 7
3 Workflow of Trassel 13
4 Trassel CLI usage description 16
5 Screenshot of code coverage report 16
6 Test-case example 18
7 Detailed workflow of Trassel 20

List of Tables

1 Evaluated testing-frameworks 9
2 Evaluated programming languages 11

Glossary

API Application Programming Interface, an interface in a software that other programs or libraries can use to utilise the functionality of the software.

Boilerplate Code that needs to be repeated with small or no difference throughout the software.

CCS Code Composer Studio, an Integrated Development Environment from Texas Instruments.

CLI Command-Line Interface, a software interface that is operated by the user through a command-shell where the software can give feedback to the user and the behaviour can be altered through parameters given on the command-line.

COM Component Object Model, a Microsoft standard for inter-process communication.

Docstring A string literal associated with a symbol. It's primarily used for documentation but can have other uses as well. The big difference to a comment is that the docstring is accessible at runtime. Examples of languages that incorporate docstrings are Python and LISP.

DSL Domain Specific Language, a language constructed for solving problems within a specific problem-domain. Examples of well known DSLs are yacc, lex, SQL, regular expressions, VHDL and LaTeX.

Dynamic testing Testing functionality of code by executing it and checking properties of the result (usually identity).

Dynamic typing A language that incorporates dynamic typing performs most of its type checking at run-time.

Finite State Machine A strategy for representing computations that depends on earlier inputs and states to handle the current input which will perhaps put the system in another state.

Firmware Software that resides in a smaller electronic device. They usually run on very limited CPUs that is sparse on both CPU time and memory usage.

FSM See Finite State Machine

Functional/integration-test A test-case that tests whether two or more software units work together as specified in the requirements.

GCC GNU Compiler Collection, a distribution of compilers for different languages and architectures.

IDE Integrated Development Environment

Object-class A class in the context of object-oriented programming.

Regression-testing Testing that is performed automatically after changes is made to the code-base to ensure that no functionality is broken by the change.

Regular expression A category of concise languages for matching patterns in text.

RPN Reverse Polish Notation (also called postfix notation) is a mathematical notation where the operator appears after all of its operands. Calculators implemented with this notation usually works by storing the operands on a stack and performing the operations on the top elements of the stack.

Static analysis Testing and verification of the code without executing it. It covers everything from simple lexical and syntactical verifiers (compiler parser, lint) to more advanced programs that analyses and mathematically proves that certain bad conditions can't occur (BLAST [1]) and analysers that detects dangerous patterns in code when comparing to some abstract pattern in a database (Coverity Prevent).

Static typing A language that incorporates static typing performs most of its type checking at compile-time (it doesn't mean that the language is type-safe however).

Testing framework The system responsible for evaluating test-cases, it also provides one or more interfaces for constructing test-cases.

Unit-test A test-case that tests a single software unit.

1 Introduction

1.1 Purpose/Objective

The purpose of this project is to develop a method for verifying the correctness of firmware for DC/DC-converters. It's part of a larger research-project at Ericsson AB called UCD3K that aims to improve the method of requirements specification, development and quality-assurance of firmware in DC/DC-converters.

The aim will be to develop a method for dynamic testing of the current code-base for the UCD3K-project. A number of unit-tests will be implemented to test the method itself and to demonstrate the solution, more specifically a couple of number conversion will be tested as well as a finite state machine.

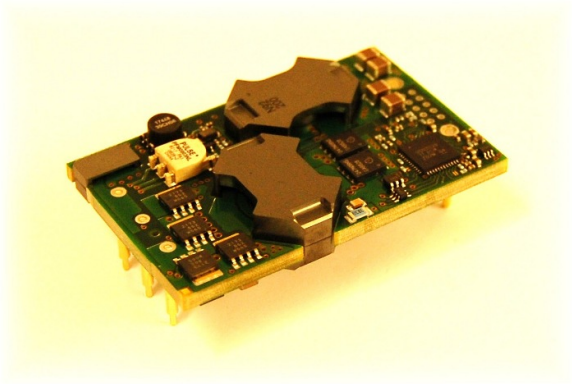


Figure 1: UCD3K DC/DC-converter

Aspects of the functionality that will be tested are results, effects and performance. The latter is an important aspect when working hard real-time applications and the idea is to be able to measure this in the unit-tests.

This project will also touch some areas of software quality not strictly ordered under software verification. For example, to test portability we first need to define its meaning and have a strategy to achieve it.

1.2 Limitations

Using testing for software verification has a very fundamental limitation which is that testing can only show that the software is incorrect, even if all the test-cases pass this doesn't mean that the software conforms to the requirement specification (unless exactly all possible uses of the software can be tested which would mean that the behaviour of the software is very trivial).

This is an important fact, we can never be 100% certain that our software is correct through testing, rather it's a balance between quality and development-

time.

Using testing instead of verification processes that deliver larger guarantees is a matter of practicality, a full proof of the software's adherence to the requirements would require a full formal specification of the operating semantics of the target language, a formal mathematical description of the requirements and an automated proof assistant software that can understand the language of the target software [2].

2 Theory

2.1 Software verification

Verification seeks to answer whether the software conforms to its given requirements which includes both functional and non-functional requirements [2].

Functional requirements is the explicit requirements given in the software requirements specification. The most common class of functional requirement deals with the software's output given some context, another functional requirement could be performance requirements of certain operations which is common in embedded and real-time systems.

Non-functional requirements deals with economical limitations or other practical issues in the solution of the problem, for example memory footprint, software security, internal documentation or code-reuse.

Software verification can be divided into two groups, static analysis and dynamic testing. Static analysis works by analysing the source-code without executing the software and finding potential flaws in the way the code is written. Dynamic testing works by executing the software or parts of it, and verifying that certain properties derived from the requirements hold. A method of dynamic testing can for example be to execute parts of a software system with different input-values and check that it gives the same output as a reference implementation that is given the same input [2].

2.2 Testing

2.2.1 Unit-testing

Unit-testing is a form of dynamic testing that aims to demonstrate the correctness of a single software-unit. Exactly what a software-unit corresponds to in the actual code depends on the situation (i.e. object-classes in object-oriented designs is a good fit for a software-unit in this context). The idea behind unit-testing is to "divide and conquer", by dividing the testing effort into smaller parts it gets easier to cover cases that might be impossible or hard to achieve efficiently while doing functional testing. For example when developing very secure systems it's common to create layers of security, if the higher-levelled security layer is correct it might very well be impossible to test for certain security-violations in the lower-levelled layer [3].

2.2.2 Testing-framework

The job of a testing-framework is to present the user with an interface to describe test-cases and takes care of the evaluation and reporting of the test-results. The user interface can take many shapes, it can be a graphical

application, a custom language or a software library for either the target language or some other high-level language.

In the case of software in embedded systems the testing-framework needs to be aware of (or adapted to) the build system(s) used to compile the target software to be able to evaluate the test-cases automatically.

2.2.3 Test coverage analysis

Coverage analysis provides the user with information of what code was executed during the evaluation of the test which can give the user an idea of what is tested and what isn't. Using this information as a guide the user can then write test-cases for the functionality not tested and increase the test coverage [2].

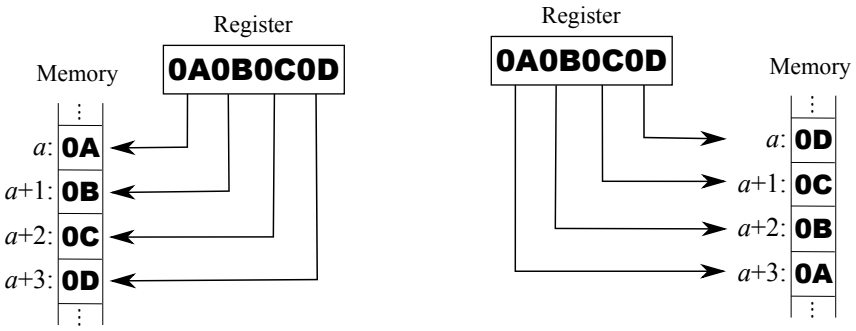
2.3 FSMs and FSM testing

Finite State Machine is a software model that describes the behaviour of a system using a finite number of states and transitions between these states based on input. When the FSM receives input a state-dependent action is performed and a state-transition is made (it can be a transition to the same state). This is a pattern that often arises when implementing different kinds of parsing, user interfaces or interaction with external devices.

For the testing of an FSM implementation to be complete it has to test all state actions and possible transitions to other states [3].

2.4 Endianness (or Byte-ordering)

Endianness is the ordering of bytes used to represent some value composed of multiple bytes and different architectures have different ways of laying out the bytes in such values which might affect the meaning of code that handles these values. In a big-endian architecture (see figure 2(b)) the least significant byte covers the highest address as opposed to little-endian (see figure 2(a)) where the least significant byte covers the lowest address [4].



(a) Big-endian architecture.

(b) Little-endian architecture.

Figure 2: Endianness

In practise endianness only matters when the code is reading memory as a different value-type than it was written as. For example, a value written as a 32bit word being read by addressing its individual bytes will behave differently on different architectures. To achieve portability while converting between different value-types it is recommended to construct an abstraction over the conversion, such as a series of functions or macros that can easily be altered to fit the current architecture [4].

2.5 Software complexity

While not directly connected to the verification process, keeping the complexity of the software system down increases quality and makes it easier to apply meaningful unit-tests.

Software complexity can be partitioned into two parts, high-level complexity which is the problem-domain of software engineering and software design and the low-level complexity which is the complexity of the written source-code (Rob Pike refers to this as microscopic complexity [5]).

Much of the low-level complexity comes from limitations inherent in the programming language. For example in C one often has to overspecify the code, embedding more information in the code than is needed to solve the problem. It can also come from bad coding practises and use of obscure language features.

One way to help keep this complexity down is to use a coding-standard that explicitly says how certain things should be written and which language features the code may use. An example of a restrictive coding-standard for embedded systems is MisraC [6] written by Misra[1]. Applying this to a software project could decrease the low-level complexity and make the source-code easier to analyse, reason about and maintain. But since MisraC

[1]Motor Industry Software Reliability Association

is so restrictive it should be considered on a per project basis, it's possible that it could increase complexity where a system is gaining from one of the forbidden language features (for example dynamic memory allocation which is forbidden in MisraC).

2.6 Domain specific embedded languages

The purpose of a DSL is to solve problems within a specific domain in a way that is more natural and efficient than trying to use a general purpose language. The idea is that the DSL models the terminology and dynamics of the problem-domain in question.

A Domain Specific Embedded Language is a DSL that is using the syntax and tools from another language (called host-language). It's implemented as a library for the host-language and the distinction between DSEL and library API is a bit fuzzy.

A large gain from constructing a DSEL is that compilers/interpreters and many tools already exists. It's also much easier to express behaviour that is normally outside of the problem-domain because you got all the libraries and functionality of the host-language for free. If the host-language is used to implement other DSELs or used by itself by the developers this might very well lower the learning curve as opposed to having a number of DSLs implemented from scratch.

The potential backside of using a DSEL rather than a DSL is that the host language imposes syntax and structure that makes the language less clear which is why languages with lighter syntax makes for more natural DSEL [7].

Stan [8] and Parsec [9] are good examples of DSELs. Stan is a HTML-templating system using Python as its host-language, it exists to make it easier to construct HTML-code from within the Python language, it also have the nice effect of catching some of the possible errors that arises when writing HTML by hand. Parsec is a language for describing parsers in the language Haskell. It operates by the idea of parser combinators, simple parsers, including higher-order parsers can be combined to form complex parsers. Parsec also demonstrates the extensibility DSELs can have, when it was implemented it was designed to work with a monadic interface [10] (that is the library interface that the developer used to combine the parser combinators). Later when applicative functors was discovered [11] it only required three rows of Haskell-code to make Parsec able to combine parsers with an applicative style as well as the monadic. Both examples makes good use of the host-language syntax and are easy to learn if the user already knows the host-language.

3 Method

3.1 Testing frameworks

Four frameworks was evaluated for use in the project. Tessy[2] which is a graphical application that integrates with the IDE and assists in the writing of simple Unit-tests, Check[3] and EmbUnit[4] which is little more than C-libraries that is compiled with the software itself and the construction of a new custom testing-framework.

Name	Tessy	Check	EmbUnit	Custom
Unit-testing	8	7	5	7
Functional-testing	no	yes	no	yes
Realtime-testing	no	no	yes	yes
Uses target architecture	yes	no	yes	yes
Extendable	1	9	4	6
Implementation-cost	4	5	4	8
Price	€4,995	0:-	0:-	0:-

Table 1: Evaluated testing-frameworks.
The scores range from 0 to and including 9, higher is better.

Tessy, while being a fairly friendly graphical application with good report generation facilities, was discovered to be very limited in the type of tests that could be written. It could not easily be scripted which means that it's hard to extend beyond whats implemented in the framework. It could only handle simple function executions and compare the result with a static value. It was for example very hard to compare a function to a reference implementation. Tessy operates by stepping through the code with a debugger and setting the test-data directly to the arguments of the tested function through the debugger as opposed to the other alternatives that operates by running static or generated C code that calls the functions that should be tested.

Check and EmbUnit is very easy to apply but is at the same time very limited in what they can achieve since the entire framework have to run inside the same run-time as the software being tested. Report generation gets very tricky to achieve, especially with EmbUnit that runs within the target hardware or on a simulator. The fact that EmbUnit runs inside a very limited environment also means that the maximum number of tests in a single build is reached fairly quick, which means that the testing process becomes tedious if it isn't automated.

The fourth alternative evaluated was to construct a new custom testing-framework that takes some of the ideas from Check and EmbUnit but auto-

[2] http://www.razorcat.de
[3] http://check.sourceforge.net
[4] http://embunit.sourceforge.net

mates the generation of test-code, can use external reference implementations, support report-generation and performing tests on multiple architectures. The downside to this alternative is that it involves a lot more work, all the named features have to be implemented from scratch during the project. Another negative aspect is that to be able to describe more expressive test-cases the framework has to provide a custom test description language which is non-trivial to design and harder to learn than the test description methods in the other alternatives.

The custom framework was chosen because it was the only alternative that provided solutions to all the posed problems.

3.2 Implementation languages

A number of languages was evaluated for implementing the testing framework and perhaps provide the basis for a test-description DSEL if it's expressive enough. If it would be infeasible to describe test-cases in a language that would either mean that a DSL has to be developed or another language would have to be used in combination with the implementation language to solve this problem.

Python [12] and Perl [13] are dynamically-typed imperative programming-languages. Perl is interpreted and Python is byte-compiled but both share the flexibility of a dynamic run-time system. Python has a light and fairly flexible syntax in addition to features like metaclasses[5] which makes it suitable as a basis for a DSEL. Both have a rich set of libraries and all of the libraries important to this project is present (most importantly COM).

Haskell [14] is a statically-typed functional programming-language with non-strict evaluation that has a very light and flexible syntax. It has a long track record of hosting DSELs and tools for handling other languages [7]. It has good performance since it's primarily a compiled language (a number of interpreters exists). The library support is lower than the other evaluated languages but it has all libraries that is important for this project. Since Haskell belongs to a programming paradigm that is fundamentally different from what the programmers at Ericsson work with the ease of adoption will probably be lower than the other alternatives.

C# [15] and Java [16] are strongly-typed imperative byte-compiled programming-languages. Both borrow much syntax and ideas from C. They both have a fairly heavy syntax and requires a lot of boilerplate code, making them unsuitable for DSELs which requires another solution for test descriptions. Java has a rich set of libraries and both have the most important libraries for this project.

None of the evaluated languages is directly bound to any proprietary software that requires purchasing a license or that limits the use of software

[5] `http://www.python.org/dev/peps/pep-0253/`

implemented in it.

Name	Python	Perl	Haskell	C#	Java
Performance	4	5	7	6	6
Maturity	8	8	6	9	9
Extendability	8	5	9	3	2
DSEL possibilities	7	3	9	1	1
Library support	8	8	5	6	8
Ease of adaption	6	4	4	8	8

Table 2: Evaluated programming languages.
The scores in the table range from 0 to and including 9, higher is better.

Python was chosen for both implementation and test description. It had the needed libraries, it's fast enough since performance is almost not an issue in this case, it's expressive enough to be usable as a DSEL and being in the imperative paradigm makes it relatively easy to adopt.

3.3 Chosen method

A custom testing framework will be developed that has the capability to interface with CCS to run tests. It will support measuring CPU cycles spent while performing a test.

The testing framework will be implemented using Python and the Python COM-libraries to communicate with the simulator and debugger. Both are mature and well supported. The Python COM-libraries is bundled with the Python distribution for Microsoft Windows.

GCC will be used to offer testing on the development machine which will be both faster and offering a possibility to test for platform independence.

The CCS IDE will be used for simulation and debugging code on the target architecture. This software is provided by Ericsson AB.

Python, all libraries and related documentation will be downloaded from the Internet.

The constructed framework will be operating by generating code from a test-case description, run this code within some environment, analyse the result and generate a report.

3.4 Criticism of chosen method

The comparisons between the frameworks and implementation languages are in many cases rough estimations and generally unscientific. This is by necessity since it would not be feasible to try to apply all the chosen frameworks or implement frameworks in all languages implement the chosen languages. This means that an alternative method could very well have been

found to be better suited to solve the problem had it just been pursued further.

Constructing a testing-framework from scratch means that code maintenance can become an issue. If the framework source-code is complex and/or badly documented it will make it very hard for the users to solve problems that might occur with the framework itself.

There is a few problems inherent with the implementation language Python. It's relatively slow, this is not an issue now but future extensions have a small risk of running into problems with it. It can be troublesome to package and distribute on Microsoft Windows platforms since it depends on the Python-runtime and a set of Python libraries. There also exists languages which is more flexible and has a lighter syntax (such as Haskell) that would make for a cleaner DSEL, neither is it the language that would be easiest for the users to adopt.

Evaluating tests by generating specific test-code instead of using fine grained control over the debugger has some drawbacks. The machine code on the target grows when testing because the test-code has to be present in the program and the build-system has to be aware of the testing-framework. Depending on how the simulator operates it might also be easier to achieve better performance measures when evaluating the tests through a debugger because there is no interfering code in the target. (This isn't the case in this project where the CCS profiling results deviates when using the debugger).

4 Realisation

4.1 Design of the Trassel testing-framework

Tests will be implemented by the user by writing test-case descriptions in a DSEL provided by Trassel. This DSEL will use Python as its host-language and incorporate ideas such as templating, dynamic application of test-data and testing against reference implementations.

The idea behind the description language is to write a code-template that will be filled with various test-data, this will allow the test-case itself to be somewhat more generic and simplify the process of constructing many similar tests.

These descriptions will be converted to a number of test-cases that will be represented internally as Python objects, they will get passed through a templating system that will apply some test-data to the code-template if needed. A test-runner will take these test-case objects, extract their test-code and compile it together with the target software, execute the target software and extract the result returned from the tests code. The test-case objects will then apply some checking function, reference implementation or expected value from the test-case description to calculate the status of the test. The report generator will go through all test-case objects and read their status to present the user with a test report as a HTML-document. See figure 7 for a more detailed graph over the workflow.

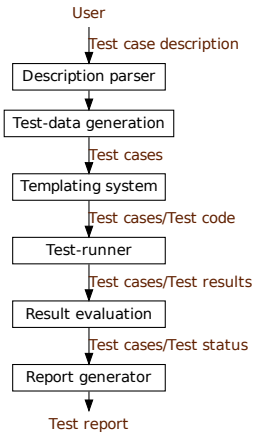


Figure 3: Workflow of Trassel

4.2 CCS proof-of-concept (Perl)

The Perl-examples from the CCS distribution was extracted and modified to work with the relevant processor model and the permission requirements to be able to control the running CCS environment was identified. Further the Perl-script was modified to run a small test-program within CCS and gather information necessary for automated testing, such as extraction of variable values and exporting profiling information.

4.3 CCS proof-of-concept (Python)

A Python CCS COM-interface was generated with the tools distributed with Python for Windows and the previous Perl-example was ported to Python statement by statement. This version also went further in interpreting the

exported profiling information so that it could extract running times for arbitrary functions.

4.4 CCS test runner

The proof-of-concept program was refactored into a module that was capable of executing an arbitrary piece of C-code and retrieve a number of variable-values from the end of the program execution and running times. A generic interface for this functionality was written as well, making it possible to switch test-environment in the code using the module.

4.5 GCC test runner

A separate test runner was developed for compiling and executing C-code with the GCC C-compiler. It implemented the same interface as the CCS test runner so that it would be simple to change between the two.

4.6 Code-generator modules

A test-case interface was designed and the test runners was modified slightly to be able to run the code contained in the test-case and fill the test-case with the results.

An object-class was written to represent a single test-case within Trassel (SingleTestCase). It contains the C-code to be tested, the results after the test runner has executed it and functionality for checking the status of the test-case.

Another object-class was constructed to contain other test-cases allowing for logical grouping of test-cases (CompositeTestCase) while still only exposing a single test-case interface.

A function was written to group the test-cases based on memory-size or a simple count so that it is possible to compile and run batches of test-cases in sequence. This aims to make it possible to run large number of tests on very limited platforms.

4.7 Embedded description-language

A couple of high-level test-case descriptions was written with expressive power, extendability and conformity with the host-language in mind. A specific description style was chosen, it was adjusted slightly to be possible to express with the syntax of Python and code for analysing it and generating test-case objects from it was implemented.

4.8 Generating reports

A HTML-report generation module was implemented using the Nevow library for generating HTML and the Pygments[6] library for highlighting snippets of source-code.

It contains one function for generating a summary of all or failed test-cases and one function for generating a detailed view of a specific test-case.

4.9 Automated testing of Trassel

Test-cases was written to verify test-case generation and code generation facilities using Python doctests and a simple script was created to evaluate them.

4.10 Test-state tree

A specialised form of the description-language was written for testing code with a lot of branching. It's designed to contain a tree of run-time states that multiple test-cases could depend on to traverse to the same run-time state every time. A simplified way to look at this functionality is to see each state as a snapshot of the memory and execution-state of the tested software.

A code generator was then written to generate the code from the state-tree to setup the run-time state for every test-case in the description.

4.11 Application interface

A user-interface was implemented using optparse distributed with Python that follows the GNU/POSIX syntax convention. It allows the users to run tests that match a certain pattern, choosing between test runners and setting rules for splitting tests into batches.

[6] `http://pygments.org`

```
Usage: runtests.py [options]

Options:
  -h, --help            show this help message and exit
  -v, --verbose         enables verbose output
  -d INT, --debug=INT   sets debug-level, default is 0
  -l, --list            lists all test descriptions and exit
  -p RE, --pattern=RE   only run tests with names matching this regular
                        expression.
  -r NAME, --runner=NAME
                        test-runner used to evaluate the tests (available
                        runners: GCC, CCS)
  -s INT, --maxsize=INT
                        sets the maximum memory-size for splitting tests into
                        batches
  -c INT, --maxcount=INT
                        sets the maximum number of tests for splitting tests
                        into batches
  --dryrun              don't actually run the tests, just generate testcode
                        and compile it
```

Figure 4: Trassel CLI usage description

4.12 Implementing code-coverage reporting

Code-coverage functionality was implemented using the gcov program from the GCC distribution. The GCC build environment was set up to compile the program with coverage reporting and a small Python program was written to run the gcov program on the files and reformat the created raw textual output to a HTML-report that shows the source-code with rows that wasn't visited highlighted.

```
            for(;;)
            {
               if(rpn.state == S_DONE)
branch 0 taken 10% (fallthrough)
branch 1 taken 90%
            break;
               if(rpn.state == S_ERROR)
branch 0 taken 0% (fallthrough)
branch 1 taken 100%
            return -1;
               tick(&rpn);
call 0 returned 100%
            }

            return pop(&rpn);
call 0 returned 100%
         }
```

Figure 5: Screenshot of code coverage report

4.13 Implementing test-cases

A couple of simple C functions together with test-cases was implemented and experimented with throughout the project to make sure that the framework solved the posed problem and to use as demonstration material.

One of the implemented C functions was a calculator using RPN notation. The expression parsing was implemented as a FSM so that it would be a useful demonstration of the test-state functionality.

5 Usage example

The following example is a simple test-case written in Trassel for the RPN calculator in appendix A.1. What the test-case does is to execute `calc` with two different expressions and check the result against a static value.

```
1  from trassel import *
2
3  class RPNTest(TestCase):
4      name = "RPN"
5
6      runners = ["GCC", "CCS"]
7
8      template = """
9  int output = calc("%s");
10 RES(&output);
11 """
12
13     format = "i"
14
15     test_simple = "1 1+"
16     expect_simple = 2
17
18     test_complex = "1 5 4+ 3 +4-5*+"
19     expect_complex = 41
```

Figure 6: Test-case example

Row 1 to 3 is mainly boilerplate code and the name "RPNTest" in row 3 has to be unique in the file. These rows doesn't really add much but is required because of the host-language.

Row 4 describes the name of the test-case that is presented to the user when running the test. This name is also used when filtering test-cases from the user interface.

Row 6 lists all compatible test-runners which stops the test-case from being evaluated if the current test-runner isn't in this list.

Row 8 to 11 contains the template of the test-code, this is the C-code that will be executed during the testing. `\%s`will be substituted with the test-data in the templating phase of the framework, this follows the simple format-string convention which is described in the user manual together with a set of other templating-systems. The `RES`macro is what the C runtime uses to pass the resulting value back to the framework.

Row 13 describes the format of the result using the Python struct lan-

guage[7]. This is used to convert the value from the C runtime to a value that can be used in the description language (and if needed it can be used to change endianness from the setting used by the test-runner).

Row 15 and 16 describes the first test. Row 15 describes the test-data that is applied to the template and row 16 describes the expected result. The name of the specific test is "simple".

Row 18 and 19 describes another test in the same way, the only difference being that the expression is a bit more complex.

[7] `http://docs.python.org/library/struct.html`

6 Result

This project has resulted in this report including the research surrounding testing-frameworks and languages suitable for constructing domain-specific languages and tools, a testing-framework (Trassel) including a documentation in the form of API-documentation, a code-coverage report generator and a user manual containing installation instructions, usage instructions and examples.

6.1 The testing-framework Trassel

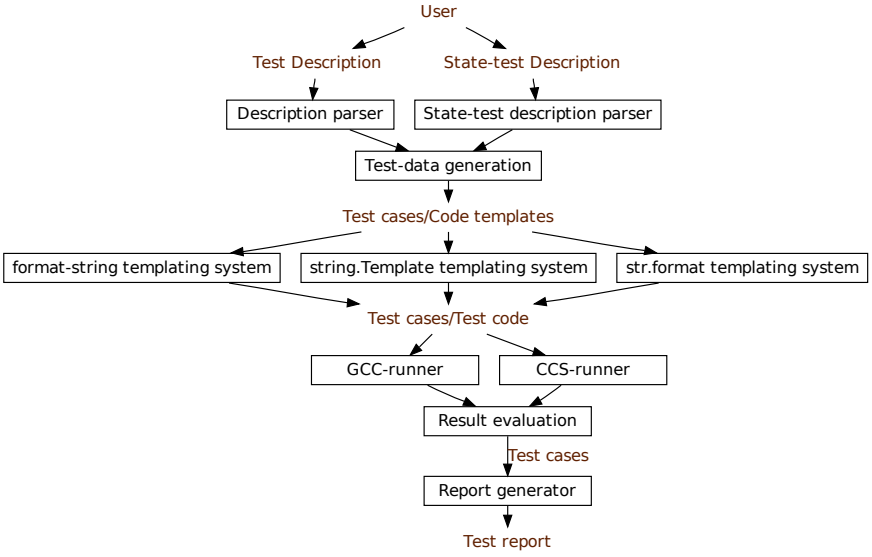


Figure 7: Detailed workflow of Trassel

6.1.1 Unit-testing

Trassel is fully capable of performing unit-tests on C code used for firmware, it's also capable of using dynamic or random test-data and testing against reference implementations.

It uses a so called test-runner to evaluate the tests and two such test-runners are implemented that supports test-case evaluation with GCC and CCS. They use a modular interface so implementing new test-runners should be fairly straight forward.

When using the GCC test-runner to evaluate a smaller number of tests the testing only takes a few seconds and it's the report-generation that takes most of this time.

Switching between GCC and CCS like this requires the software to be portable with respect to among other things, endianness and available data-types.

It features a module for generating HTML reports from the evaluated tests. It generates overview reports for all tests, failed tests and detailed reports per test.

It's invoked through a CLI that conforms to the conventional GNU/POSIX syntax and it supports a number of flags to control the behaviour of the test-evaluation (for example the ability to filter test-cases with a regular expression).

6.1.2 Description language

A language for describing test-cases was built upon the Python language. It builds on the syntax for classes in Python without extending this in a major way, placing it inside the grey-zone between DSEL and Python-library.

Using Python classes this way created a bit of boilerplate because of syntax required for the language to be valid Python-code. The language itself is very powerful, it automates the generation of tests and Python-code can be used almost everywhere if needed which enables the user to use any Python-library to solve problems. The implementation is quite small and simple which makes it easy to extend. This was put to the test during the project when an extension to the language was written to handle test-states to avoid repeating of initialisation-code for groups of tests that form a tree-structure.

6.1.3 Code coverage

Functionality to perform code coverage analysis when evaluating the tests was implemented but wasn't integrated with the rest of the Trassel framework. Because the functionality is currently only working with gcov test-coverage reports can only be obtained when using the GCC test-runner.

6.1.4 Documentation

Documentation for the Trassel framework include a user manual [17] and API-documentation.

The user manual describes how to set up Trassel to work with a code-base and how to invoke the framework through the CLI.

The API-documentation is written in docstrings in the source-code and a software called epydoc is used to collect this information and generate a HTML-formatted document.

7 Analysis / Discussion

7.1 Automated unit-testing

There is little doubt that software development including firmware development can benefit from automated testing. A large set functionality can be tested in a relatively small amount of time after the code has changed, making it easy to detect unwanted behaviour of the code after a feature is added or a bug is fixed (This is also called regression-testing).

A positive effect of doing unit-testing on smaller software units is that the test-data can, in many cases be simpler and smaller which means that it's easier to get a high coverage of the functionality. As an example, a function that only takes 4 boolean values as parameters would be possible to test for all possible inputs, giving the verification a 100% certainty.

The developed testing-framework is fully capable of performing automated testing and it has been demonstrated that this can be done effectively throughout the development process thanks to the fast GCC test-runner and the possibility of running only a selected set of tests with the user interface.

Switching to GCC for testing also forces the user to test the portability of the target software, if endianness isn't handled correctly or unportable data-types or language features are used these issues will surface quickly when the user switches between the test-runners.

Having the ability to do unit-testing is not enough to ensure the quality of the software however. There has to be some form of consensus or policy for when to write test-cases and what to write test-cases for. For software projects with high quality requirements nothing less than 100% test-coverage might be acceptable.

7.2 Python

Using Python for implementing the framework turned out well. Python made it easy to construct prototypes and experiment with parts of the functionality that was unclear, for example the CCS interaction and the description language.

Using libraries for source-code highlighting (Pygments), HTML rendering (Nevow/stan) and document formatting (docutils) made it easy to quickly try different ideas with the report generation.

7.3 DSEL

Implementing the test-case description language as a DSEL embedded in Python demonstrated several aspects of DSELs from the referenced material. It was certainly easier to implement in comparison to a custom DSL, the Python code handling the description language is just around 200 lines of

code (comments and docstrings included), implementation of interpreters and libraries for a custom DSL would probably be larger by orders of magnitude.

Having access to a full general purpose language in the description language together with all the libraries available for it also demonstrates a strength of DSELs, the user is much less likely to run into a wall where the wanted behaviour simply can't be expressed by the language. This follows the philosophy of "Making simple things easy and hard things possible".

7.4 Criticism/limitations

7.4.1 Limitations inherent in testing

As mentioned in the introduction testing has rather strong limitations that developers needs to be aware of. Unless all possible combinations of test-data can be tested there is no way to be 100% certain that the code fully adheres to the requirements.

It's also very hard to reason about how meaningful the tests are. If tests doesn't cover parts of a function it's easy to realise that it isn't tested but the reverse isn't necessarily true. A line of code evaluated during the test doesn't mean that the test checked the functionality of that line and even if it did it might only have partially checked the lines functionality, some important aspects can easily be overlooked.

7.4.2 Testing method

The method that Trassel use to evaluate test-cases is to generate C-code which is then compiled together with the target software and executed. An alternative would be to execute the target software with a debugger and use the debugger to set up scenarios to be tested. This wasn't evaluated because a testing framework that operated in this way was thought to be more complex to implement. It would render the test-case descriptions quite different to the currently implemented in Trassel but since no attempts was made to implement debugger-driven testing it's hard to do any meaningful comparisons.

7.4.3 Testing with different platforms

If portability isn't needed or wanted switching between GCC and CCS might become an awkward technique since a lot of functionality might not work under one of the platforms and this might not be possible to solve by compensating for it in the test-cases.

Even if portability is a wanted goal, setting up the build-environments to use the same source-code files to make swapping between them a simple procedure can be complicated. Just maintaining multiple build-systems for

the source code means that the user have to do a bit of extra work throughout the development process.

7.4.4 Test-case description language

Implementing the description language by embedding it in Python did give some overhead due to syntax requirements to be valid Python and boilerplate code. This is something that is very hard to improve without abandoning Python as a host-language.

The syntax used to describe test-data and optional result checking functionality for that test-data is grouped by simply having the same name. This makes the language more fragile to typos and harder for the user to interpret than a language that would use more explicit groupings (see the example under "Usage examples").

The notation used when writing the test-code is a bit verbose because of the existing templating systems and the `RES` macro used to pass the result to the framework, it is thinkable that there exists a solution that makes the descriptions look clearer and is simpler to work with.

8 Conclusion

8.1 Testing

Testing is an important part of software verification today and this is equally true for firmware development as well. Writing and evaluating test-cases in the form of regression-tests, that is performing testing of code continuously throughout the development process decreases the time it takes to find bugs and errors in the code.

Testing firmware code presents some interesting problems since it's hard to perform unit-tests on the actual hardware. The Trassel framework developed during this project solved this problem by offering the possibility of executing code either on a simulator in the CCS environment or by running the code on the development environment using GCC to compile the relevant parts of the source-code.

The Trassel framework doesn't solve all testing related problems, it merely helps the user to automate the evaluation of the test-cases. The user still has to setup the environments and build-systems to compile and run the test-code. If some of this work is discovered to be possible to automate it might be a good idea to extend Trassel to cover that as well. Trassel is implemented in a modular fashion for this reason, it's supposed to be easy to add new description languages, templating systems, test-runners or test-case concepts.

8.2 DSELs

Developing tools and DSLs in modern high-level languages is a sound strategy since the gain can be very high and they are often relatively simple and fast to implement. The largest problem with this strategy is that a good DSEL design can be hard to achieve and that it puts a lot of pressure on the language to be expressive and have a reasonable light syntax. The first issue is of course true of DSLs in an even higher degree, invalidating this issue if a new tool or language is needed regardless of implementation method. The language requirements issues is more interesting though, of the languages that was evaluated in this report only Haskell and Python was deemed to be a fitting host-language and a quick look at DSELs in the real world seems to confirm this. Haskell has been used to construct a large number of DSELs as mentioned in [7] and Python certainly has a number of libraries which can be said to be DSELs (stan from Nevow [8] for example).

8.3 Where to go from here

During the development of Trassel a number of features was thought of that wasn't implemented and several fields was found that wasn't researched. Some of the more important of these are listed here to act as a help how to improve Trassel and the knowledge of firmware verification.

8.3.1 Integrate implemented code coverage function

The implemented code coverage reporting functionality wasn't integrated with the rest of the framework due to lack of time. Having this functionality in the Trassel framework would greatly aid the developer in deciding what functionality to write test-cases for.

A suggested feature together with this is to have an overview of the coverage in all source files with summations on directories. This wouldn't be hard to implement and would simplify finding untested parts of the code.

8.3.2 Packaging to ease installation and distribution

Currently Trassel uses a number of third-party libraries to solve different problems and many of these have different and sometimes hard to perform installation methods. Some dependencies might be possible to eliminate, but it will still be a good idea to package Trassel together with the libraries it depend on so that only the Python runtime, this Trassel package and perhaps cygwin (for GCC) needs to be installed.

8.3.3 Improving the description language

The description language could be improved in a number of ways. It might be best to use the current language for a while and get a good feeling for which areas are important but when that is known these are a few of the alternatives.

- using nested Python-classes for associating result checking with the specific test-data instead of depending on the specific naming.
- cleverer templating system that eliminates the need for the `RES`-macro or at least hides it so that the test-code looks clearer.
- reducing the number of templating systems and description languages. Having a separate test-state description language might be an unnecessary burden for the user.

8.3.4 Robustness features

When the compiler or the C runtime fails for some reason the error isn't handled by Trassel at all. In the case of the CCS-runner this means that the CCS application shows the error and in the case of GCC the error is shown in the command-shell where Trassel was invoked. It's not clear how much would be gained if Trassel could handle and report such failures itself, one reason it would be a good idea for Trassel to detect errors is to perform binary searches among the test-cases to find out which test-case is causing problems. Whether such a function is needed will turn out after more use.

8.3.5 Evaluation of alternatives for formal verification

A lot of research is done in the field of formal verification of software today and there exists a number of tools and documents about it. In places where formal verification can be done testing simply can't compete, formal verification can show that a requirement holds for all cases where testing only can show correctness for a tiny fraction of all possible cases.

Technologies of interest:

- Cryptol - http://www.cryptol.net
- Isabelle - http://www.cl.cam.ac.uk/research/hvg/Isabelle
- ACL2 - http://www.cs.utexas.edu/ moore/acl2
- Coq - http://coq.inria.fr

References

[1] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transfer*, 2005.

[2] The Motor Industry Software Reliability Association. Report 6: Verification and validation. Technical report, February 1995.

[3] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley, 2003.

[4] Intel. Endianness white paper. Technical report, November 2004.

[5] Rob Pike. Notes on programming in c, February 1989.

[6] The Motor Industry Software Reliability Association. Misra-c:2004. Technical report, 2007.

[7] Paul Hudak. Modular domain specific languages and tools. In *IN PROCEEDINGS OF FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE*, pages 134–142. IEEE Computer Society Press, 1998.

[8] Nevow. `http://divmod.org`.

[9] Daan Leijen University, Daan Leijen, and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[10] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[11] Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18, 2007.

[12] Python. `http://python.org`.

[13] Perl. `http://perl.org`.

[14] Haskell. `http://haskell.org`.

[15] C#. `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

[16] Java. `http://www.sun.com/java`.

[17] Trassel user manual, 2009.

A Appendices

A.1 RPN calculator source-code

A.1.1 rpn.h

```
#ifndef __RPN_H
#define __RPN_H

typedef enum RPNState RPNState;
enum RPNState {
    S_MAIN,
    S_READING,
    S_DONE,
    S_ERROR
};

typedef struct RPN RPN;
struct RPN {
    char *input;
    RPNState state;
    int sp;
    int stack[100];
};

void push(RPN *rpn, int n);
int pop(RPN *rpn);
void tick(RPN *rpn);
int calc(char *expr);

#endif /* __RPN_H */
```

A.1.2 rpn.c

```
#include <ctype.h>
#include "rpn.h"

void push(RPN *rpn, int n)
{
    rpn->stack[rpn->sp++] = n;
}

int pop(RPN *rpn)
{
    return rpn->stack[--rpn->sp];
}

void tick(RPN *rpn)
{
    char i = *rpn->input;

    switch(rpn->state)
    {
    case S_MAIN:
        if(isdigit(i))
        {
            push(rpn, 0);
            rpn->state = S_READING;
            return;
        }
        if(i == '+')
```

```
            push(rpn, pop(rpn) + pop(rpn));
        else if(i == '-')
        {
            int sub = pop(rpn);
            push(rpn, pop(rpn) - sub);
        }
        else if(i == '*')
            push(rpn, pop(rpn) * pop(rpn));
        else if(i != ' ')
        {
            rpn->state = (i == '\0') ? S_DONE : S_ERROR;
            return;
        }
        rpn->input++;
        break;
    case S_READING:
        if(isdigit(i))
        {
            push(rpn, pop(rpn)*10 + (int)(i - '0'));
            rpn->input++;
        }
        else
            rpn->state = S_MAIN;
        break;
    case S_DONE:
        break;
    case S_ERROR:
        break;
    }
}

int calc(char *expr)
{
    RPN rpn;

    rpn.sp = 0;
    rpn.state = S_MAIN;
    rpn.input = expr;

    for(;;)
    {
        if(rpn.state == S_DONE)
            break;
        if(rpn.state == S_ERROR)
            return -1;
        tick(&rpn);
    }

    return pop(&rpn);
}
```